Problem Set 2

yl2nr

data structures and algorithms 2 - fall 2023

due september 28, 2023 at 11:59 pm

Collaboration Policy: You are encouraged to collaborate with up to 4 other students, but all work submitted must be your own *independently* written solution. List the computing ids of all of your collaborators in the `collabs` command at the top of the tex file. Do not share written notes, documents (including Google docs, Overleaf docs, discussion notes, PDFs), or code. Do not seek published or online solutions for any assignments. If you use any published or online resources (which may not include solutions) when completing this assignment, be sure to cite them. Do not submit a solution that you are unable to explain orally to a member of the course staff. Any solutions that share similar text/code will be considered in breach of this policy. Please refer to the syllabus for a complete description of the collaboration policy.

Collaborators: list your collaborators

Sources: list your sources

problem 1 *Solving Recurrences*

Prove a (as tight as possible) O (big-Oh) asymptotic bound on the following recurrences. You may use any base cases you'd like.

1. For the following two recurrences, it may be helpful to draw out the tree. However, you should prove the asymptotic bound using induction.
   - $T(n) = T(\frac{n}{2}) + T(\frac{n}{4}) + T(\frac{n}{8}) + n$ https://www.overleaf.com/project/6505d047a5f98e3f98c5d7c9
     Solution:

     Proof. Want to show that the function $T(n) = T(n/2) + T(n/4) + T(n/8) + n \in O(n \log_2 n)$.
     For our base case, consider $n = 1$, where $T(1) = 1$, a constant.
     Now, let's assume that for some positive integer k, $T(n) \in O(n \log_2 n)$ for all n up to 2^k.
     Next, we want to prove that $T(n) \in O(n \log_2 n)$ for $n = 2^{k+1}$.
     Using the recurrence relation:

     $$T(2^{k+1}) = T(2^k) + T(2^{k-2}) + T(2^{k-3}) + 2^k$$

     By our induction hypothesis, $T(2^k) \in O(2^k \log_2 2^k) = O(k \cdot 2^k)$.
     Similarly, $T(2^{k-2}) \in O((k-2) \cdot 2^{k-2})$, and $T(2^{k-3}) \in O((k-3) \cdot 2^{k-3})$.
     So, we can rewrite:

     $$T(2^{k+1}) \leq m \cdot k \cdot 2^{k+2} + 2^k$$

     for some $m > 0$.

     $$T(2^{k+1}) \leq m \cdot 2^{(k+1)} \cdot 2^{k+2} + 2^k$$

     Choosing $c = 2m$, we get:

     $$T(2^{k+1}) \leq 2m \cdot (2^{k+1} \cdot (k+2))$$

     Since $2m$ is a constant multiplier, we can say:

     $$T(2^{k+1}) \in (O(2^{k+1} \log_2 2^{k+1})$$

     Therefore, by induction, we've shown that $T(n) \in O(n \log_2 n)$ for all positive integers n. $\square$

- $T(n) = 2T(\frac{n}{3}) + T(\frac{n}{6}) + n$

  Solution:

  Proof. We aim to show that the function $T(n) = 2T(n/3) + T(n/6) + n$ has a time complexity of $O(n \log_3 n)$.
  For the base case, when $n = 1$, $T(1) = 1$, which is a constant.
  Assume that for some positive integer k, $T(n)$ is $O(n \log_3 n)$ for all n up to 3^k.
  We want to prove that $T(n)$ is $O(n \log_3 n)$ for $n = 3^{k+1}$.
  Using the recurrence relation:

  $$T(3^{k+1}) = 2T(3^k) + T(3^{k-1}) + 3^{k+1}$$

  By our induction hypothesis, $T(3^k)$ is $O(3^k \log_3 3^k) = O(k \cdot 3^k$.
  Similarly, $T(3^{k-1})$ is $O((k-1) \cdot 3^{k-1})$.
  So, we can rewrite:

  $$T(3^{k+1}) \leq m \cdot k \cdot 3^{k+2} + 3^{k+1}$$

  for some $m > 0$.

  $$T(3^{k+1}) \leq m \cdot 3^{(k+1)} \cdot 3^{k+2} + 3^{k+1}$$

  Choosing $c = 3m$, we get:

  $$T(3^{k+1}) \leq 3m \cdot (3^{k+1} \cdot (k+2))$$

  Since $3m$ is a constant multiplier, we can say:

  $$T(3^{k+1}) \in O(3^{k+1} \log_3 3^{k+1})$$

  Therefore, by induction, we've shown that $T(n)$ is $O(n \log_3 n)$ for all positive integers n. □

2. For the following recurrence relations, indicate: (i) which case of the Master Theorem applies (if any); (ii) justification for why that case applies (if one does) i.e., what is a, $f(n)$, ε, etc; (iii) the asymptotic growth of the recurrence (if any case applies).

   - $T(n) = 7T(\frac{n}{5}) + n \log n$

     Solution:

     (i) Case 1 applies
     (ii)
     $a = 7$, $b = 5$, $\delta = \log_5 7 > 1$
     $f(n) = n \log n$, $f(n) \in O(n^{\log_5 7 - \epsilon})$, for any $\epsilon < (\log_5 7 - 1)$
     (iii)
     $T(n) \in \Theta(n^{\log_5 7})$

- $T(n) = 3T(\frac{n}{3}) + n \log n$

  Solution:

  (i)
  No case applies.
  (ii)
  $a = 3, b = 3, \delta = \log_3 3 = 1$
  $f(n) = n \log n, \forall \epsilon > 0, f(n) \notin O(n^{\delta - \epsilon}), f(n) \notin \Omega(n^{\delta + \epsilon}$, and $f(n) \notin \Theta(n^{\delta})$

PROBLEM 2 *Climate History*

Scientists call *paleoclimatologists* study the history of climate on earth before instruments were invented to measure temperatures, precipitation, etc. They try to reconstruct climate history using data found from analyzing rocks, sediments, tree rings, fossils, ice sheets, etc.

A group is trying to better understand periods of low precipitation (e.g. dryness) for a time period that spans many thousands of years. Using various data sources, they have assigned a value d_i for the relative dryness for each time-unit (let's say it's measured for every century) for this very long time period. For each time-unit i, one of five values has been assigned as d_i:

- -3 for very high precipitation
- -1 for high precipitation
- 0 for normal (or unknown)
- 1 for low precipitation
- 3 for very low precipitation

The scientists want to find the score for the driest period of consecutive years in their data. Because the effects of a drought are cumulative, the score for a period starting at time i and ending at time j will be the value when all scores from i and j are added. Consider this example with $n = 16$:

d_i	0	3	3	0	-3	-3	1	0	3	-1	-1	0	1	1	3	0
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The period [1:2] looks high with a score of 6. But the highest score is 7 for this period [6:14]. Note there are some negative values in the middle of this period, but the high values around those make it worth having a period that includes those negative values. (The period [6:15] also has score 7. There can be more than one period with the largest score, but you just need to return the highest score, so the existence of multiple periods with the same best score doesn't matter.)

There are many ways to solve this problem, including a $\Theta(n^2)$ brute-force approach that calculates scores for all combinations of starting and ending values for a period and keeps track of the largest score. You need to do better than that. (Though if you were ever to code your solution, coding the brute-force approach is a good way to test your algorithm. Coding is **not** required for this problem.)

What you need to do: Describe an algorithm (clearly in words or in pseudocode) that uses a divide and conquer algorithm that solves this problem in $\Theta(n \log n)$ time. Your description should make it clear what the base case is, what work is done in dividing and in combining, and what recursive subproblems are solved. Along with your algorithm's description, give a brief but convincing argument that the time complexity is $\Theta(n \log n)$.

The inputs to your algorithm will be n, the number of samples, and a list d that stores n values, where each is either -3, -1, 0, 1 or 3. Your algorithm will output the largest dryness score for a period in the data (as described above).

Solution:

To look for the highest dryness score over a period, we can apply the idea of divide and conquer to recursively divide the array of all days into subarrays of single days. At each subarray, we calculate the dryness score. Then we combine those subarrays while comparing dryness scores on each side and also the sum of dryness scores from the midpoint to both sides by all possible length, so that we can keep track of the highest dryness score.

Divide

1. Split the input list (array) into two halves: *left_half* and *right_half*.

2. Recursively split each half into another two halves and denote the midpoint position at each split.

Conquer

1. When the array is split into base cases, which means only one element is in the subarray, then return the value of value of the element, dryness score, in the subarray.
2. For each subarray, we get a max dryness score inside that array, denoted as $MaxDryness$. For the base case, $MaxDryness =$ element in the array.
3. For non-base cases, $MaxDryness = max(MaxDryness$ of $LeftOfSub_a$, $MaxDryness$ of $RightOfSub_a$, $MaxJointDryness$ of $sub_a)$, which will be calculated during Combine.

Combine

1. When we combine two subarrays to one longer subarray sub_a, we find the maximum dryness score $MaxJointDryness$ for sub_a that crosses the midpoint of sub_a as joined by two subarrays, $LeftOfSub_a$ and $RightOfSub_a$ to combine.
2. Let variable $LeftDrynessScoreMax$ be $-INT\_MAX$. We start computing the maximum sum of dryness scores of continuous days by moving from the midpoint to left by one element at a step and compare the current sum to the $LeftDrynessScoreMax$. We update $LeftDrynessScoreMax$ to be the larger one of the two values (current sum and $LeftDrynessScoreMax$).
3. Repeat the same step for the right half subarray so we get $RightDrynessScoreMax$.
4. We add $LeftDrynessScoreMax$ and $RightDrynessScoreMax$ and get the $MaxJointDryness$. Then for each subarray sub_a, we return the $MaxDryness$ as the max value among ($MaxDryness$ of $LeftOfSub_a$, $MaxDryness$ of $RightOfSub_a$, $MaxJointDryness$ of sub_a).
5. The final combine back to the original array will produce the maximum of dryness score of any possible periods.

The time complexity of this algorithm is $O(n \log n)$ as by dividing, the recursion tree has depth of $\log n$. Each conquer and combine step is linear so is at most n. Thus the final time complexity is $O(n \log n)$.

PROBLEM 3 *Fast Transformations*

Computer graphics software typically represents points in n dimensions as $(n+1)$-dimensional vectors. To make transformations on the points (e.g., rotating a modelled figure, zooming in, or making the figure appear as though seen through a fish-eye lens), we use a $(n+1) \times (n+1)$ matrix T which defines the transformation, and then we multiply each vector by this matrix to transform that point. That is, for vector v, the transformed vector is $v' = T \times v$.

Let's say we are developing software for very high dimension graphics (n dimensions), and we have a transformation T that we would like to apply to a particular point n times. Develop an algorithm which can multiply this $(n+1)$-dimensional point by T (the $(n+1) \times (n+1)$ transformation matrix) n times in $o(n^3)$ (little-oh of n^3) time. Prove this run time.

Solution:

The idea is to apply divide and conquer so that we divide the total multiplication in half and recursively calculate matrix $T^{2k\frac{n}{2^k}}$, until the base case we have $(T^n)^1$, then we times vector v.

```
def transformation (T,v,n):
    if n == 1:
        return T × v
    if n % 2 == 0:
        return transformation(TT,v,n/2)
    else:
        return T × transformation(TT,v,n/2)
```

As the calculation of the product of matrices at each case is strictly less than n^3 if we apply Strassen's algorithm to calculate, and there are only $\log n$ recursive cases, the final time complexity is $o(n^3)$.

PROBLEM 4 *Receding Airlines*

You have been hired to plan the flights for Professor Floryan's brand new passenger air company, "Receding Airlines." Your objective is to provide service to n major cities within North America. The catch is that this airline will only fly you East.

You recognize that in order to enable all your passengers to travel from any city to any other city (to the East) with a single flight requires $\Omega(n^2)$ different routes. Prof. Floryan says that the airline cannot be profitable when supporting so many routes. Another option would be to order the cities in a list (from West to East), and have flights that go from the city at index i, to the city at index $i+1$. This requires $\Theta(n)$ routes, but would mean that some passengers would require $\Omega(n)$ connections to get to their destination.

1. Devise a compromise set of routes which requires no passenger have more than a single connection (i.e. must take at most two flights), and requires no more than $O(n \log n)$ routes. Prove that your set of routes satisfies these requirements.

2. After a few years, passengers start demanding routes from East to West, and you decide to support new routes from East to West (in addition to supporting routes from West to East). Show that with routes in *both* directions, it is possible to connect all n cities with just $O(n)$ routes such that no passenger needs more than a single connection to get to their destination.

Note: In class so far, we've used recurrence relations to count an algorithm's time complexity, i.e., how many basic operations are executed. Here we're using one to count the number of flights. While this is a bit different than measuring an algorithm's time-complexity, we can still use a divide and conquer approach and a recurrence relation to answer these questions.

Solution:

1. The idea is to build a binary tree with the root of the middle airport. So that every airport on the left side of the root node or inner node is always on the west side of its parent. All the airports on the right are always on the east side of parents.

   Then all the branches in the tree will be routes, and we also add routes from the root to every leaf of the tree.

   Thus at every level, the airport on that level will have routes to all the airports on all the lower levels.

   Assume we have n airport, then from the top 1 airport (the one in the middle) will have $n-1$ routes. We can then apply the Divide and Conquer idea to recursively calculate the number of routes on each lower level until we reach the leaves.

$$T(n) = 2T(\frac{n}{2}) + (n-1) \tag{1}$$

$$T(1) = 0 \tag{2}$$

   To prove this idea meets the requirement that flying from any airport to another one takes at most one transition, we can take any two airports A_i and A_j. Assume A_i is on the west side of A_j, so we need to fly from A_i to A_j. Then we just find the least common ancestor node (airport) of A_i and A_j, which can be noted as A_m. Then according to the design, we can definitely fly from A_j to A_m, and then transfer another flight from A_m to A_j, as both A_i and A_j are on the lower levels of A_m.

   To prove the time complexity is $O(n \log n)$, we can apply the Master Theorem.

$$a = 2, b = 2, \delta = \log_2 2 = 1, \qquad f(n) = n - 1 \in \Theta(n^\delta) = \Theta(n)$$

Thus, this falls into Case 2, leads to the implication that

$$T(n) \in \Theta(n \log n) \implies T(n) \in O(n \log n)$$

As $T(n)$ is the total number of flights, total number of flights is $O(n \log n)$.

2. As we want to take at most one transition flight from any airport to any other airport, we then need a hub airport to connect with all other airport bidirectionally. So from any airport we can first fly to the hub airport and then transfer to another flight to the destination airport.

   So using the same set-up in part 1, we can let root airport to be the hub. Then at each level, we just need to connect the node to the root in two directions, so taking routes of 2. Then we can recursively calculate the total number of routes.

$$T(n) = 2T(\frac{n}{2}) + 2 \tag{3}$$

   We have proved that this design ensures that from any airport to any other airport takes up to 1 transition, 2 flights.

   To prove that the total number of routes is $O(n)$, we apply Master Theorem.

$$a = 2, b = 2, \delta = \log_2 2 = 1, \qquad f(n) = 2 \in O(n^{\delta - 0.1}) = O(n^{0.9})$$

   Thus, the falls in to case 1.

$$T(n) \in \Theta(n) \implies T(n) \in O(n)$$

   As $T(n)$ is the total number of flights, total number of flights is $O(n)$.

PROBLEM 5 *Bazinga!*

Theoretical Physicist Sheldon Cooper has decided to give up on String Theory in favor of researching Dark Matter. Unfortunately, his grant-funded position at Caltech is dependent on his continued work in String Theory, so he must search elsewhere. He applies and receives offers from MIT and Harvard. While money is no object to Sheldon, he wants to ensure he's paid fairly and that his offers are at least the median salary among the two schools' Physics departments. Therefore, he hires you to find the median salary across the two departments. Each school mantains a database of all of the salaries for that particular school, but there is no central database.

Each school has given you the ability to access their particular data by executing *queries*. For each query, you provide a particular database with a value k such that $1 \leq k \leq n$, and the database returns to you the k^{th} smallest salary in that school's Physics department.

You may assume that: each school has exactly n physicists (i.e. $2n$ total physicists across both schools), every salary is unique (i.e. no two physicists, regardless of school, have the same salary), and we define the *median* as the n^{th} highest salary across both schools.

1. Design an algorithm that finds the median salary across both schools in $\Theta(\log(n))$ total queries.
2. State the complete recurrence for your algorithm. You may put your $f(n)$ in big-theta notation. Show that the solution for your recurrence is $\Theta(\log(n))$.
3. Prove that your algorithm above finds the correct answer. *Hint: Do induction on the size of the input.*

Solution:

1. The idea is to start at the highest in one data set and lowest in the other one, and if set 1 has a higher median than set 2, then we query the $n + \frac{n}{2}$ of set1 and $n - \frac{n}{2}$ of set2. Then we recursively compare the current query value of two sets and make new queries with positions as (current position $+\frac{n}{2^i}$) for the set with a lower value and (current position $-\frac{n}{2^i}$) for the set with a higher value, where i is the number of time of the query. The base case would be when $\frac{n}{2^i} = 1$, then we just return the mean of two current queries as the median.

```
def findMean(position1 = n, position2 = 0, num_salary = n):
    if length_level <= 1:
        query1 = query_set1(position1)
        query2 = query_set2(position2)
        return (query1+query2)/2
    query1 = query_set1(position1)
    query2 = query_set2(position2)
    if query1 > query2:
        return findMean(position1 - num_salary/2, position2 + num_salary/2, num_salary/2)
    else:
        return findMean(position1 + num_salary/2, position2 - num_salary/2, num_salary/2)
```

2. The recurrence has the following mathematical relation:

$$T(n) = T(\frac{n}{2}) + 2$$

On each level, we have a cost of 2, as we are doing one query with a cost of 1 on each of the two sets. Therefore, $f(n) \in O(\log n)$.

As we have totally $\log n$ levels of the recurrence tree, and on each level, we only have one case, the total time is $T(n) = 2 \log n$.

Thus, $T(n) \in \Theta(\log n)$.

3. *Proof.*

Proceed by Induction. We first check the base case num_salary $n = 1$. When n = 1, there are two salary levels in total, the function returns the mean of two levels as the median, which is correct. So the base case is true.

Then we assume that $\forall n < k$, findMean(n,0,n) is correct. We now consider num_salary n = k.

findMean(k, 0, k) returns findMean(k/2, k/2, k/2). While as n = k/2 $<$ k, findMean(k/2, k/2, k/2) is correct, so findMean(k, 0, k) is also correct.

Thus, as the base case is true, and when case with n is true, case with n+1 is also true, the algorithm is true to be return the correct answer. □

PROBLEM 6 *Mission Impossible*

As the newly-appointed Secretary of the Impossible Missions Force (IMF), you have been tasked with identifying the double agents that have infiltrated your ranks. There are currently n agents in your organization, and luckily, you know that the majority of them (i.e., strictly more than $n/2$ agents) are loyal to the IMF. All of your agents know who is loyal and who is a double agent. Your plan for identifying the double agents is to pair them up and ask each agent to identify whether the other is a double agent or not. Agents loyal to your organization will always answer honestly while double agents can answer arbitrarily. The list of potential responses are listed below:

Agent 001	Agent 002	Implication
"002 is a double agent"	"001 is a double agent"	At least one is a double agent
"002 is a double agent"	"001 is loyal"	At least one is a double agent
"002 is loyal"	"001 is a double agent"	At least one is a double agent
"002 is loyal"	"001 is loyal"	Both are loyal or both are double agents

1. A group of n agents is "acceptable" for a mission if a majority of them ($> n/2$) are loyal. Suppose we have an "acceptable" group of n agents. Describe an algorithm that has the following properties:
   - Uses at most $\lfloor n/2 \rfloor$ pairwise tests between agents.
   - Outputs a smaller "acceptable" group of agents of size at most $\lceil n/2 \rceil$.
2. Using your approach from Part 1, devise an algorithm that identifies which agents are loyal and which are double agents using $\Theta(n)$ pairwise tests.

Solution:

1. The idea is to divide the whole list of agents into pairs of two (one single agent left out if n is odd). Then we will have exactly $\lfloor \frac{n}{2} \rfloor$ pairs for pairwise comparison. As we know there are more loyal agents in the big group, we are sure that for all agents who are said to be loyal, at least a half of them are really loyal.

   Then for each pair (a,b) comparison result

   (a) If agent a is loyal according to b, then no matter what b is labeled, we add a to the smaller group set 1.

   (b) If a is double according to b, then we do nothing.

   (c) If agent b is loyal according to a, then no matter what a is labeled, we add b to the smaller group set 2.

   (d) If b is double according to a, then we do nothing.

   Finally, we choose the smaller set between set 1 and set 2, and we add the single agent left out in the pairing if n is odd to the set. We return the set as the smaller acceptable group of agents.

2. The idea is that we recursively run the algorithm in part 1 on the smaller acceptable group generated in each level, and we stop until we only have one loyal agent left.

   Then we let this one loyal agent *Agentk* pair up with all the rest agents, so n-1 pairwise tests in total. Those said to be loyal by *Agentk* are truly loyal and those said to be double agents by *Agentk* are truly double agents.

   The time complexity of the algorithm is $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + ... + 1 + (n-1) = 2n - 1 \in \Theta(n)$.